# Multi Robot Collaboration in Grid-Based Environments Using Multi-Agent Reinforcement Learning (MARL)

Fadime Yaren Durmuş 200201003

Mustafa Kağan Aytaç 200202026

Instructor: Asst. Prof. Deniz Gençağa

29.05.2025

https://github.com/FadimeYaren/multi-robot-collaboration-using-deep-learning-RoboticsSpring2025ABU

## 1 Introduction

### What is a Multi-Robot System?

A multi-robot system is a system in which multiple robots communicate and collaborate simultaneously to accomplish various tasks. The robots are trained beforehand to adapt to the environment and conditions. They learn to find the optimal path to complete the task in the most efficient and fastest way possible, and then begin practical execution accordingly.

### What is a Single-Robot System?

A single-robot system consists of one robot operating independently, typically under the control of a centralized system.

### What are the Differences Between Multi-Robot and Single-Robot Systems?

The main difference lies in communication and control. Single-robot systems are usually managed by a central controller, whereas multi-robot systems may follow different management strategies depending on the application. While single robots handle individual tasks, multi-robot systems involve complex network structures to coordinate and complete tasks collaboratively.

## 2 Background and Related Work

2.1 Coordination Architectures in Multi-Robot Systems

There are three different coordination architectures: Centralized, Decentralized, and Distributed.

In Centralized systems, robots are managed from a single control point, making the system simple.

In Distributed systems, robots make their own decisions. They communicate with other robots in the environment, exchange information, and complete the task within a complex network structure.

In Decentralized systems, robots only consider nearby robots and act accordingly. They do not have access to global information.

2.2 Approaches for Multi-Robot Collaboration

- **Rule-based systems**
  Fixed task allocation, logical rules, sequential processing
- **Optimization-based methods**
  Task allocation using Hungarian Algorithm, Linear Programming
- **Auction-based mechanisms**
  Contract Net Protocol, decentralized task bidding
- **Heuristic approaches**
  Genetic algorithms, swarm optimization
- **Machine Learning-based methods**
  Supervised, imitation, and reinforcement learning
- **Graph-based methods**
  GNNs for communication structure, influence modeling

Among these, **reinforcement learning** has gained significant attention due to its ability to learn from interaction without explicit programming.

### What is Reinforcement Learning and why is it used in robotics?

Reinforcement Learning is a type of learning method that aims to discover the most efficient long-term strategy.

Reinforcement Learning (RL) is a machine learning system that enables an agent (such as a robot) to learn how to behave in an environment through experience. The agent receives a reward after each action and tries to learn the best behavior based on these rewards. The goal is to maximize cumulative reward through trial and error, eventually developing a strategy (policy).

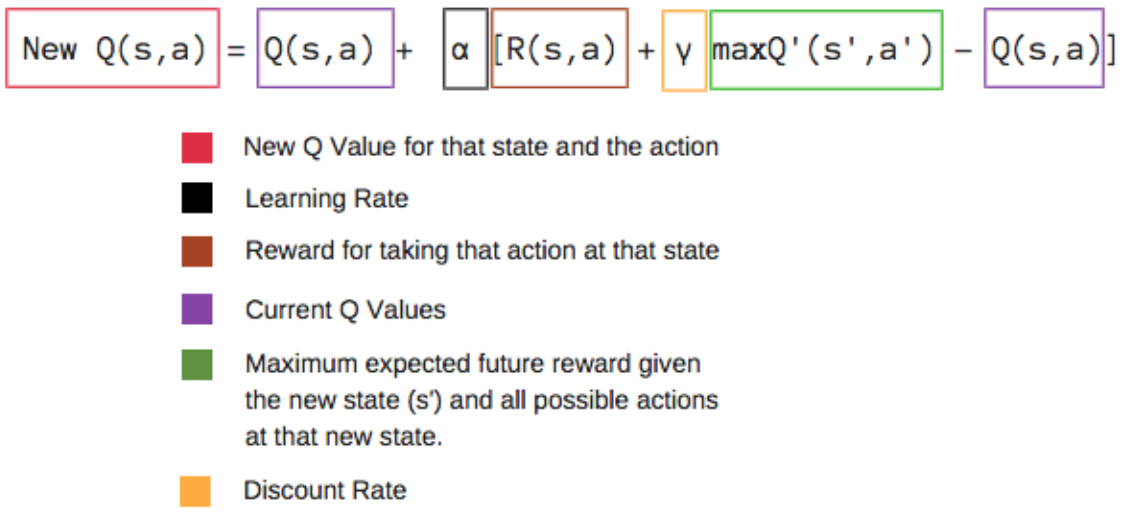
The general logic behind RL is “do the right thing, earn the reward.”

2.4 RL-Based Learning Algorithms

| Algorithm Description | Algorithm Description |
| --- | --- |
| **Q-learning** | The agent learns which action is better through a Q-table. |
| **SARSA** | Similar to Q-learning, but allows for more flexible decision-making. |
| **Deep Q-Network (DQN)** | A combination of Q-learning and deep learning (neural networks). |
| **Policy Gradient** | The agent learns the policy directly; no Q-table is used. |

2.4.1 Q-Learning

Q-Learning enables robots to learn Q-values, which are action-value functions, in order to maximize long-term rewards.



In this simulation, the Q-Learning algorithm has been used. In short, Q-Learning works as follows:

The robot gains rewards by completing tasks in the environment through trial and error. Based on the rewards it receives (i.e., the tasks it completes), it builds a Q-table for the environment. This Q-table is updated using a backpropagation-like method. Depending on its current state, the robot uses the continuously updated Q-table to determine which path is likely to yield better results. With the final version of the Q-table, the robot determines a policy, which becomes the optimal path—i.e., the fastest and most efficient route to complete the task.

**Agent**: The entity that makes decisions and takes actions (e.g., a robot).
**Environment**: The world in which the agent operates (e.g., a maze, a game area).
**State**: The agent's current position or condition at a given moment.
**Action**: The possible moves or operations the agent can perform.
**Reward**: A numerical value (positive or negative) the agent receives based on its action.
**Policy**: The strategy that defines which action the agent takes in a given state.
**Q-Value**: The expected reward of taking a specific action in a specific state.

```
Episode 1 | Reached Goal: False
Step 0: 0 → ↑ → 0, Reward: -0.5
Step 1: 0 → ↓ → 4, Reward: 0.0
Step 2: 4 → ← → 4, Reward: -0.5
Step 3: 4 → ↓ → 8, Reward: 0.0
Step 4: 8 → ← → 8, Reward: -0.5
Step 5: 8 → ↓ → 12, Reward: -1
```

$$Q(0,\uparrow)=0+0.3\cdot(-0.5-0)=-0.15$$

$$Q(0,\downarrow)=0+0.3\cdot(0+0.99\cdot\max(Q(4))-0)=0$$

$$Q(4,\leftarrow)=0+0.3\cdot(-0.5-0)=-0.15$$

$$Q(4,\downarrow)=0+0.3\cdot(0+0.99\cdot\max(Q(8))-0)=0$$

$$Q(8,\rightarrow)=0+0.3\cdot(-0.5-0)=-0.15$$

$$Q(8,\downarrow)=0+0.3\cdot(-1-0)=-0.3$$

For Episode 1:

The robot selects an action completely at random. The chosen action (↑) is upward, and since this action moves the robot outside the grid boundaries, it receives a penalty. As a result, the Q-values are calculated immediately, and the state-action pair that caused the penalty is updated in the Q-table. Then, the robot proceeds with other actions. It appears that the robot did not reach any reward during Episode 1, and by the end of the

episode, it generates a Q-table as shown below. This process continues similarly for each episode.

| States | Left | Bottom | Right | Up |
|---|---|---|---|---|
| 0 | -0.15 | 0 | 0 | -0.15 |
| 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 |
| 4 | -0.15 | 0 | -0.3 | 0 |
| 5 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 |
| 8 | -0.15 | -0.3 | 0 | 0 |

For Episode 23:

The robot takes its actions based on the Q-table it has developed up until Episode 23. As seen in Step 7, it reaches the reward. Consequently, state S14 receives a positive value. Since the reward is located at S15, the robot learns that it must definitely move from S14 to S15, and from this point on, it consistently chooses the rightward (→) action to reach S15 from S14. The updated Q-table begins to receive positive feedback values due to the received reward, and once the learning is complete, the optimal path is formed.

$Q(0,\uparrow) = -0.26 + 0.3 \times (-0.5 + 0.99 \times \max(Q(0)) - (-0.26))$ than
$Q(0,\uparrow) = -0.26 + 0.3 \times (-0.5 + 0 - (-0.26)) = -0.26 + 0.3 \times (-0.24) = -0.33$

$Q(0,\leftarrow) = -0.46 + 0.3 \times (-0.5 + 0.99 \times \max(Q(0)) - (-0.46))$ than
$Q(0,\leftarrow) = -0.46 + 0.3 \times (-0.5 + 0 - (-0.46)) = -0.46 + 0.3 \times (-0.04) = -0.47$

$Q(0,\downarrow) = 0 + 0.3 \times (0 + 0.99 \times \max(Q(4)) - 0)$ than $\max(Q(4)) = \max(-0.42, 0, -0.94, 0) = 0$ than
$Q(0,\downarrow) = 0 + 0.3 \times (0 + 0 - 0) = 0$

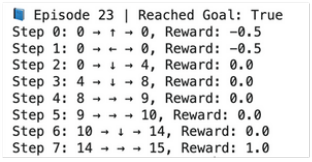
$Q(4,\downarrow) = 0 + 0.3 \times (0 + 0.99 \times \max(Q(8)) - 0)$ than $\max(Q(8)) = \max(-0.47, -0.94, 0, 0) = 0$ than $Q(4,\downarrow) = 0$

$Q(8,\rightarrow) = 0 + 0.3 \times (0 + 0.99 \times \max(Q(9)) - 0)$ than $\max(Q(9)) = \max(0, 0, 0, -0.51) = 0$ than $Q(8,\rightarrow) = 0$

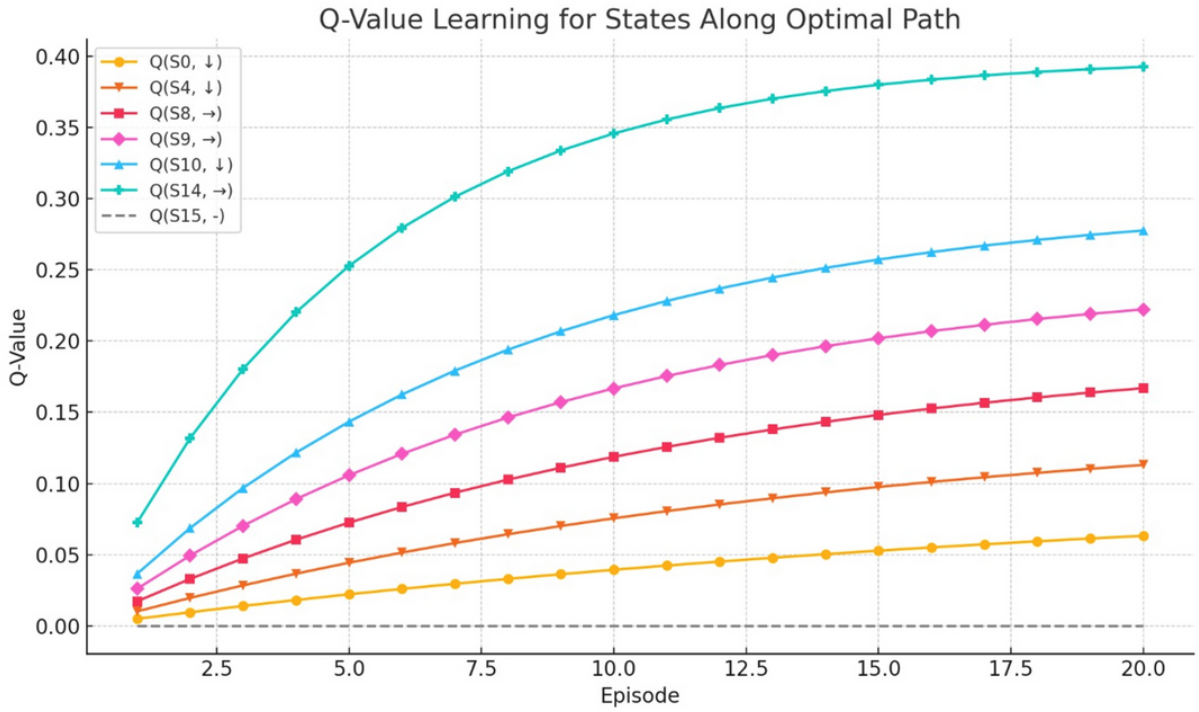
$Q(9,\rightarrow) = 0 + 0.3 \times (0 + 0.99 \times \max(Q(10)) - 0)$ than $\max(Q(10)) = \max(0, 0, -0.30, 0) = 0$ than $Q(9,\rightarrow) = 0$

$Q(10,\downarrow) = 0 + 0.3 \times (0 + 0.99 \times \max(Q(14)) - 0)$ than $\max(Q(14)) = \max(0, 0, 0.30, 0) = 0.30$ than
$Q(10,\downarrow) = 0.3 \times (0.99 \times 0.30) = 0.3 \times 0.297 = 0.09$

$Q(14,\rightarrow) = 0.30 + 0.3 \times (1 + 0.99 \times \max(Q(15)) - 0.30)$ than $\max(Q(15)) = 0$ than
$Q(14,\rightarrow) = 0.30 + 0.3 \times (1 - 0.30) = 0.30 + 0.21 = 0.51$



As can be seen from the table below, the usage percentage of S14 is very high, indicating it is part of the optimal path. Right after that, we see the paths leading to S14, such as from S10 and then from S9.




The Q-Table prepared for the optimal path in the final state of the learned robots is as follows.

| State | ← | ↓ | → | ↑ |
|---|---|---|---|---|
| 0 | 0.44 | 0.95 | 0.95 | 0.44 |
| 1 | 0.94 | -1.00 | 0.96 | 0.45 |
| 2 | 0.95 | 0.97 | 0.95 | 0.46 |
| 3 | 0.96 | -0.99 | 0.45 | 0.45 |
| 4 | 0.45 | 0.96 | -1.00 | 0.94 |

| | | | | |
|---|---|---|---|---|
| 5 | 0.00 | 0.00 | 0.00 | 0.00 |
| 6 | -1.00 | 0.98 | -1.00 | 0.96 |
| 7 | 0.00 | 0.00 | 0.00 | 0.00 |
| 8 | 0.46 | -1.00 | 0.97 | 0.95 |
| 9 | 0.96 | 0.98 | 0.98 | -1.00 |
| 10 | 0.97 | 0.99 | -1.00 | 0.97 |
| 11 | 0.00 | 0.00 | 0.00 | 0.00 |
| 12 | 0.00 | 0.00 | 0.00 | 0.00 |
| 13 | -1.00 | 0.48 | 0.99 | 0.97 |
| 14 | 0.98 | 0.49 | 1.00 | 0.98 |
| 15 | 0.00 | 0.00 | 0.00 | 0.00 |

### 2.4.2 Deep Q-Network (DQN)

It is the version of the classic Q-Learning algorithm combined with artificial neural networks. Instead of a Q-table, it uses a Deep Neural Network to estimate Q-values for each state and action. In short, it is a reinforcement learning algorithm that uses deep learning to compute Q-values.

| | |
|---|---|
| Neural Network | Girdi: durum (state), Çıktı: her aksiyon için tahmini Q-değeri |
| Replay Buffer | Deneyimler (state, action, reward, next_state) hafızada saklanır ve rastgele örneklenir (overfitting'i azaltır) |
| Target Network | Ana ağın sabit bir kopyasıdır. Kararlılığı artırır. Belirli aralıklarla güncellenir. |
| ε-greedy policy | Keşif (exploration) ve sömürü (exploitation) dengesi sağlar. |

#### 2.4.3 Centralized Training with Decentralized Execution (CTDE)

During the training phase, the robots are trained together and learn collectively. This includes building a shared Q-table and defining a joint policy. In the real world, when the robots begin to operate practically, they make decisions independently, adapt to the environment, observe other robots, and adjust their behavior accordingly. There is no need for real-time communication.

**Training Phase (Centralized):**
• Robot A and Robot B are trained together.
• They are aware of each other's actions.
• They learn as a team using a shared reward function.

**Real-Time Operation (Decentralized):**
• Robot A makes decisions based solely on its own cameras and sensors.
• Robot B behaves independently in the same way.
• Even without communication, they work in harmony thanks to the strategies learned during training.

#### 2.4.4 Multi-Agent Algorithms

| Method | Action Space | Coordination | Communication | Advantage |
|---|---|---|---|---|
| IQL | Discrete | None | None | Simple, independent |
| MADDPG | Continuous | Yes | Centralized (training) | Continuous action spaces |
| QMIX | Discrete | Team Q-value | None | Centralized value decomposition |
| MAAC | Discrete/Cont. | Yes | Indirect via attention | Effective information selection |
| CommNet | Usually cont. | Yes | Learnable | Shared communication |
| TarMAC | Usually cont. | Yes | Selective messaging | Targeted message passing |
| GNN-Based | Mixed | Yes | Via graph structure | Learns dynamic relationships |

**What are the relevant open sources and how have they contributed?**

| Category | Tool / Platform | Description |
|---|---|---|
| Simulation | Gazebo + ROS | Physical robot and sensor simulations |
| | PettingZoo | MARL environments (games, task allocation) |
| RL Library | RLlib (Ray) | Distributed and multi-agent RL training |
| | PyMARL | Designed for MARL algorithms like QMIX |
| MARL Environment | MPE (Particle Env) | Simple physical multi-agent environments |
| | SMAC (StarCraft Challenge) | Strategy-focused MARL benchmark environment |
| Communication-Based | CommNet / TarMAC | Coordination through learnable communication |
| Real Robot System | Robotarium | Remotely assign tasks to real robots |

## 3 Environment Design and Simulation

**How simulation environment designed?**

This project focuses on designing a simulated environment in which one or more robots can collaboratively prepare a hamburger by interacting with different stations in a grid-based kitchen. The main purpose of the system is to enable these robots to navigate, collect, process, and combine ingredients to produce a valid hamburger.

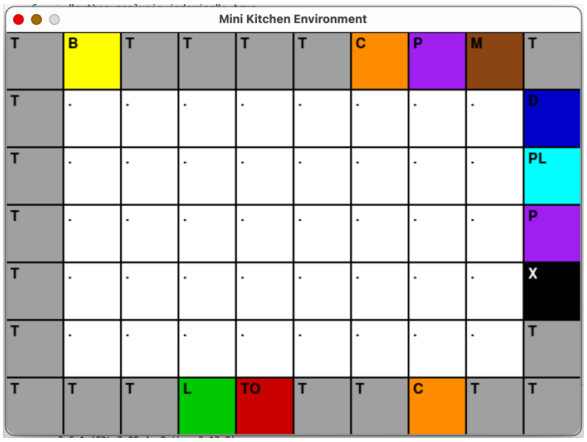
The environment consists of a fixed-size 2D 7X8 grid where each cell can represent either a walkable area, a station, or an object.

### Grid Layout and Station Design

Each cell on the grid corresponds to a specific entity:

- B: Bread Station
- L: Lettuce Station
- TO: Tomato Station
- C: Cutting Board
- P: Pan (Stove)
- M: Meat Station
- PL: Plate Station
- T: Table (work surfaces where items can be dropped and combined)
- D: Delivery Station
- X: Trash Bin
- .: Walkable free space

These stations were initially planned in a spreadsheet and later implemented in Python using a matrix-based system. Each type of station supports specific interactions; chopping vegetables on the cutting board or cooking meat on the pan.




### Items

Another important point of the environment is merging items. Assets and combination rules are defined one by one. The rule ise, you can merge items only if there is a plate.

Combinations:

single combinations

Plate + Bread --> plate_bread.png

Plate + Tomato (Chopped) --> plate_tomato.png

Plate + Lettuce (Chopped) --> plate_lettuce.png

Plate + Meat (Cooked) --> plate_meat.png

binary combinations

Plate + Bread + Tomato --> plate_bread_tomato.png

Plate + Bread + Lettuce --> plate_bread_lettuce.png

Plate + Bread + Meat --> plate_bread_meat.png

Plate + Tomato + Lettuce --> plate_tomato_lettuce.png

Plate + Tomato + Meat --> plate_tomato_meat.png

Plate + Lettuce + Meat --> plate_lettuce_meat.png

triple combinations

Plate + Bread + Tomato + Lettuce --> plate_bread_tomato_lettuce.png

Plate + Bread + Tomato + Meat --> plate_bread_tomato_meat.png

Plate + Bread + Lettuce + Meat --> plate_bread_lettuce_meat.png

Plate + Tomato + Lettuce + Meat --> plate_tomato_lettuce_meat.png

quadruple combination

Plate + Bread + Tomato + Lettuce + Meat (Full Burger) --> plate_burger.png

There are other functions of this project as LIVE/REPLY modes. In the Live mode user controls our chef robot manually. Additionally, Reply for runing episodes which are ran during in the learning stage of agents.

**Why was hamburger production chosen?**
Hamburger production seemed suitable for simulating real-life task division. Although the main focus of this project was creating a grid-based environment, it has the potential to be extended by

integrating MARL agents into a Unity-based multiplayer game called *Kitchen Chaos*, which could be open-sourced in the future. Targeting multiplayer games is highly appropriate for simulating multi-agent collaboration. All that would be needed is to replace the players with agents.

**Infrastructure for Logging and Metrics**

Logging is the process of recording all actions performed by agents in the simulation (e.g., movement, interactions, object handling) and any changes in the environment. These records typically include timestamp, agent ID, position, action, and interaction details.

By analyzing what agents do in which states, their learning process can be evaluated. In multi-agent systems, each agent's contribution can be measured. In the long term, the same scenario can be replayed with different algorithms for comparison.

**Agent Logs (agent_logs)**
Each agent's actions are logged in detail:

- Action type: pickup, drop, chop, cook, merge, discard
- Position: The grid position where the action took place
- Time: Time step measured by pygame.time.get_ticks()
- Details: Object carried, transformed ingredient, etc.

**Environment Logs (burger_logs)**
Plate creation and combination events triggered by agent interactions are recorded:

- Plate position
- Contents (ingredients)
- Which agent(s) contributed
- Final plate type (e.g., plate_burger, plate_bread_meat, etc.)

**Snapshot Logs (snapshot_logs)**
For each time step:

- Agents' positions
- Direction (agent_dirs)
- Held item (if any)
- Status of all objects in the environment (e.g., a tomato on a table)

**Possible Performance Metrics**

Total Reward:
Represents the cumulative reward collected by each agent throughout training. For example, a successful hamburger delivery yields a positive reward added to this metric.

Task Completion Rate:
Indicates how many hamburgers are successfully completed per episode. It directly reflects the system's effectiveness.

Collaboration Score:
Measures whether multiple agents contributed to the creation of a single hamburger and how much each agent contributed to the process.

Ineffective Action Rate:
Tracks the ratio of actions that were unnecessary or had no useful outcome (e.g., moving randomly, picking up incorrect items).

Action Efficiency:
Calculates the average number of actions required to complete a hamburger. Lower values indicate more efficient behavior.

Convergence (Learning Stability):
Shows when agents begin to consistently perform well during training. It reflects the point at which their behavior stabilizes and converges toward an optimal strategy.

Versioning Mini Kitchen During Development

Version 00–03: Core Structure and Interface

- Created the initial grid layout with symbolic station representations (e.g., B for Bread Station, T for Table).
- Added a movable agent with directional control and basic animations.
- Implemented a side information panel to display agent state.

Version 04–06: Item Interaction and Basic Object Handling

- Introduced the ability for agents to pick up and drop items.
- Defined item states (e.g., {"type": "meat", "state": "raw"}) and contextual interactions using the space key.
- Enabled agents to place items on tables and visualized objects in the environment.

Version 07–08: Cooking, Chopping, and Visual Feedback

- Implemented logic for chopping and cooking ingredients at specific stations.
- Replaced symbolic item representation with dynamically loaded PNG images for each item state.

Version 09–10: Ingredient Combination and Burger Construction

- Developed a merging system to combine plate ingredients into full hamburgers with one-to-one mappings (e.g., Plate + Meat → Plate_Meat).
- Standardized item types and combination rules to ensure consistency and prevent logical conflicts.
- Ensured that once combined, ingredients cannot be separated (realistic cooking constraint).

Version 11: Multi-Agent Logging and RL Readiness

- Introduced full multi-agent support (NUM_AGENTS) with independent position, inventory, and action logs.
- Implemented three-tier logging: agent_logs for actions, burger_logs for plate combinations, and snapshot_logs for full environment states.
- Established the infrastructure required for reward tracking, collaboration analysis, and replay visualization.
- Laid the foundation for MARL experiments (e.g., DQN, CTDE).

Version 12–13: Replay Mode and Live/Replay Toggle

- Enabled users to replay past episodes using a GUI episode selector.
- Implemented Live vs. Replay toggling and resolved agent desync issues during replay.

Agents can perform the following primitive actions:

- Move: Up, Down, Left, Right
- Interact: Pick up, drop, chop, cook, combine, or discard items depending on the context

**How Does Collaboration Take Place?**

In this study, robots (agents) are designed to collaborate within the same environment to produce hamburgers. Each agent has its own inventory, position, and perception. However:

- **Different agents can contribute to the same plate.**
  For example, Agent 0 can cook and place the meat, while Agent 1 can add the tomato.
- This collaboration is realized through **contributions toward shared goals** (e.g., completing the same hamburger).
- Since agents work toward shared tasks, cooperation emerges via **uncoordinated but complementary actions**, which is especially suitable for Centralized Training with Decentralized Execution (CTDE) scenarios.

**How Is the Reward System Designed?**

The reward function is designed to encourage agents to learn meaningful and efficient behaviors. The proposed structure is as follows:

| Event | Reward |
|---|---|
| Correct ingredient picked up | +1 |
| Ingredient correctly chopped/cooked | +2 |
| Ingredient properly added to plate | +3 |
| Hamburger completed | +10 |
| Incorrect action (e.g., discarding item) | -2 |
| Useless movement | -0.1 |

Rewards are assigned individually to agents. However, the overall success (e.g., completing a hamburger) is the result of teamwork, so team contribution is indirectly rewarded.

## 6 Experiments and Results

Firstly, I tried to determine end of the episode as a making complete burger. But it meant you were teaching a baby to run before she could crawl. So, even in the first episode, it couldn't finished the episode even though time reached 55 minutes.

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `i
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `i
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

✦ Executing (55m 6s)

But then end of the episode determined as interacting with any element. So reaching reward was easier. There is some results for the agents reaching to goal:

```
Episode 1/3 — Rewards: [-0.6, -0.4]
Episode 2/3 — Rewards: [-0.4, -0.6]
Episode 3/3 — Rewards: [0.7, 0.0]
```

```
Episode 1/10 — Rewards: [-0.2, -0.5]
Episode 2/10 — Rewards: [-0.6, 0.5]
Episode 3/10 — Rewards: [1.0, -0.1]
Episode 4/10 — Rewards: [0.8, -0.30000000000000004]
Episode 5/10 — Rewards: [-0.8999999999999999, -0.1]
Episode 6/10 — Rewards: [0.8, -0.7999999999999999]
Episode 7/10 — Rewards: [1.0, 0.0]
Episode 8/10 — Rewards: [0.8, 0.0]
Episode 9/10 — Rewards: [1.0, -0.30000000000000004]
Episode 10/10 — Rewards: [1.0, -0.1]
```

According to our study, every movement made in the mini kitchen is designed to be recorded, including the direction of the agents, the location of the items they hold, and the dynamic changes in the environment. Even if our study does not reach the desired result, it forms an original and gradually developed basis for reporting.

In the future our plan is running it with more agents and we will determine step by step goals. So, baby can learn crawl than walking than running. Also applying different algortihms, approaches and training agents for up level of this mini game are our other options.

## 7 References


Testa, A., Carnevale, G., & Notarstefano, G. (2023). *A Tutorial on Distributed Optimization for Cooperative Robotics: from Setups and Algorithms to Toolboxes and Research Directions*. arXiv preprint arXiv:2309.04257. https://doi.org/10.48550/arXiv.2309.04257

Samii, S. M., Su, J., & Brown, M. (2024). *Multi-Agent Reinforcement Learning: A Review of Challenges and Applications in Robotics*. arXiv preprint arXiv:2408.11822. https://doi.org/10.48550/arXiv.2408.11822